The C Programming Language
Part 1

Goals of this Lecture
Help you learn about:
• The decisions that were made by the designers* of C
• Why they made those decisions
• The fundamentals of C

Why?
• Learning the design rationale of the C language provides a richer understanding of C itself
• A power programmer knows both the programming language and its design rationale

* Dennis Ritchie, then later, members of standardization committees

Historical context - 1972
Operating systems were programmed in assembly language (i.e., in machine instructions)
Efficient, expressive, easy to translate to machine language, but not portable from one computer instruction set to another; hard to write programs, hard to debug, maintain...

Application programs were in "high-level" languages such as Algol, COBOL, PL/1, (newly invented) Pascal
Goals of these languages: Ease of programming, expressiveness, structured programming, safety, data structures, portability
Not fully achieved: safety, expressiveness, portability
Not even attempted: modularity

Goals for C language - 1972
Program operating-systems in a "high-level" language
Need: ease of programming, (reasonable) expressiveness, structured programming, data structures, modularity, compilable on a 64-kilobyte computer
Don’t even attempt: safety
When possible, have a bit of: portability

Goals for Java language - 1995
Program operating-systems in a "high-level" language
Need: ease of programming, (reasonable) expressiveness, structured programming, data structures, modularity, safety, portability, automatic memory management
It’s not that Java was particularly innovative (in these respects). By 1995, decades of computer-science research had made it straightforward to achieve all these goals at once. In 1972, nobody knew how.
Goals of C

<table>
<thead>
<tr>
<th>Designers wanted C to:</th>
<th>But also:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support system programming</td>
<td>Support application programming</td>
</tr>
<tr>
<td>Be low-level</td>
<td>Be portable</td>
</tr>
<tr>
<td>Run fast</td>
<td>Be portable</td>
</tr>
<tr>
<td>Be easy for people to handle</td>
<td>Be easy for computers to handle</td>
</tr>
</tbody>
</table>

Conflicting goals on multiple dimensions!

Agenda

- Data Types
- Operators
- Statements
- I/O Facilities

Primitive Data Types

- integer data types
- floating-point data types
- no character data type (use small integer types instead)
- no character string data type (use arrays of small ints instead)
- no logical or boolean data types (use integers instead)

Integer Data Types

- integer data types: char, short, int, long
  - char is 1 byte
  - Number of bits per byte is unspecified! (but in the 21st century, pretty safe to assume it's 8)
  - sizes of other integer type is not fully specified but constrained:
    - int is natural word size
    - $2 \leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)}$

On CourseLab

- Natural word size: 4 bytes (but not really!)
- char: 1 byte
- short: 2 bytes
- int: 4 bytes
- long: 8 bytes

Integer Literals

- Decimal: 123
- Octal: 0173 = 123
- Hexadecimal: 0x7B = 123
- Use "L" suffix to indicate long literal
- No suffix to indicate short literal, instead must use cast

Examples

- int: 123, 0173, 0x7B
- long: 123L, 0173L, 0x7BL
- short: (short)123, (short)0173, (short)0x7B

Unsigned Integer Data Types

Both signed and unsigned integer data types
Unsigned Integer Data Types

Both signed and unsigned integer data types

- signed integer types: int, short, long
- unsigned integer types: unsigned char, unsigned short, unsigned int, and unsigned long
- char might mean signed char or unsigned char
- Define conversion rules for mixed-type expressions
  - Generally, mixing signed and unsigned converts signed to unsigned
  - See King book Section 7.4 for details

What decisions did the designers of Java make?

Unsigned Integer Literals

Decisions

- Default is signed
- Use "U" suffix to indicate unsigned literal

Examples

- unsigned int:
  - 123U, 0173U, 0x7BU
  - 123, 0173, 0x7B will work just fine in practice; technically there is an implicit cast from signed to unsigned, but in these cases it shouldn't make a difference.
- unsigned long:
  - 123UL, 0173UL, 0x7BUL
- unsigned short:
  - (unsigned short)123, (unsigned short)0173, (unsigned short)0x7B

Signed and Unsigned Integer Literals

The rules:

<table>
<thead>
<tr>
<th>Literal</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>dd...d</td>
<td>int</td>
</tr>
<tr>
<td>0dd...d</td>
<td>int</td>
</tr>
<tr>
<td>0dd...dU</td>
<td>unsigned int</td>
</tr>
<tr>
<td>dd...dU</td>
<td>unsigned int</td>
</tr>
<tr>
<td>dd...dL</td>
<td>long</td>
</tr>
<tr>
<td>0dd...dL</td>
<td>long</td>
</tr>
<tr>
<td>dd...dUL</td>
<td>unsigned long</td>
</tr>
<tr>
<td>0dd...dUL</td>
<td>unsigned long</td>
</tr>
</tbody>
</table>

The type is the first one that can represent the literal without overflow

Character Data Types

Back in 1972, some computers had 6-bit bytes, some had 7-bit bytes, some had 8-bit bytes; the C language had to accommodate all these

By 1985, pretty much all computers had 8-bit bytes

- The ASCII character code fits in 7 bits
- One character per byte
- It would be a very strange 21st-century C compiler that supported other than 8-bit bytes

The C character type

- char can hold an ASCII character
- char might be signed or unsigned, but since 0 ≤ ASCII ≤ 127 it doesn't really matter
- If you're using these for arithmetic, you might care to specify signed char or unsigned char

Character Literals

- single quote syntax: 'a'
- Use backslash (the escape character) to express special characters

Examples (with numeric equivalents in ASCII):

- 'a': the character (97, 01100001, 61)
- '\\': the character, octal character form
- '\\': the character, hexadecimal character form
- 'b': the character (98, 01100010, 62)
- 'A': the & character (65, 01000001, 41)
- 'B': the & character (66, 01000010, 42)
- 'O': the null character (0, 00000000, 0)
- 'Z': the zero character (48, 00110000, 30)
- 'A': the one character (49, 00110001, 31)
- 'n': the newline character (10, 00001010, 3A)
- 't': the horizontal tab character (9, 00001001, 39)
- '\': the backslash character (92, 01011100, 5C)
- '"': the single quote character (96, 01100000, 60)

Strings and String Literals

Issue: How should C represent strings and string literals?

Rationale:

- Natural to represent a string as a sequence of contiguous characters
- How to know where char sequence ends?
- Store length before char sequence?
- Store special "sentinel" char after char sequence?
Strings and String Literals

Decisions
- Adopt a convention
  - String is a sequence of contiguous chars
  - String is terminated with null char ("\0")
  - Use double-quote syntax (e.g. "hello") to represent a string literal
  - Provide no other language features for handling strings
  - Delegate string handling to standard library functions

Examples
- "a" is a char literal
- "abcd" is a string literal
- "a" is a string literal

Unicode and UTF-8

Back in 1970s, English was the only language in the world, so we only needed this alphabet:

ASCII: American Standard Code for Information Interchange

In the 21st century, it turns out that there are other people and languages out there, so we need:

- Unicode and UTF-8 encoding of Unicode
  - Solution: UTF-8 encoding of Unicode
    (This won’t be on the exam…)

Logical Data Types

- no logical or Boolean data type
- Represent logical data using type char
  - Or any integer type
  - Or any primitive type!!!
- Convention: 0 ⇒ FALSE, ≠0 ⇒ TRUE
- Convention used by:
  - Relational operators (<, >, etc.)
  - Logical operators (!, &&, ||)
  - Statements (if, while, etc.)

Aside: Logical Data Type Shortcuts

Note
- Using integer data to represent logical data permits shortcuts

```java
int i;
if (i) /* same as (i != 0) */
    statement1;
else
    statement2;
```

Aside: Logical Data Type Dangers

Note
- The lack of logical data type hampers compiler’s ability to detect some errors with certainty

```java
int i;
i = 0;
if (i = 5)
    statement1;
```

What happens in Java?

What happens in C?
Floating-Point Data Types

Back in 1972, each brand of computer had a different (and slightly incompatible) representation of floating-point numbers. This was standardized in 1985; now practically all computers use the IEEE 754 Floating Point standard, designed by Prof. William Kahan of the Univ. of California at Berkeley.

- three floating-point data types: float, double, and long double
- sizes unspecified, but constrained:
  \[\text{sizeof(float)} \leq \text{sizeof(double)} \leq \text{sizeof(long double)}\]

On CourseLab (and on pretty much any 21st-century computer)
- float: 4 bytes
- double: 8 bytes
- long double: 16 bytes

Floating-Point Literals

- fixed-point or "scientific" notation
- Any literal that contains decimal point or "E" is floating-point
- The default floating-point type is double
- Append "F" to indicate float
- Append "L" to indicate long double

Examples
- double: 123.456, 1E-2, -1.23456E4
- float: 123.456F, 1E-2F, -1.23456E4F
- long double: 123.456L, 1E-2L, -1.23456E4L

Data Types Summary: C vs. Java

Java only
- boolean, byte

C only
- unsigned char, unsigned short, unsigned int, unsigned long

Sizes
- Java: Sizes of all types are specified, and portable
- C: Sizes of all types except char are system-dependent

Type char
- Java: char is 2 bytes (to hold all 1995-era Unicode values)
- C: char is 1 byte

Continued next lecture

Agenda

Data Types
Operators
Statements
I/O Facilities